Lecture 15: Concolic Testing

17-355/17-665/17-819: Program Analysis

Rohan Padhye

October 30, 2025

* Course materials developed with Jonathan Aldrich and Claire Le Goues

Recap: Symbolic Execution

```
1  int x=0, y=0, z=0;
2  if(a) {
3    x = -2;
4  }
5  if (b < 5) {
6    if (!a && c) { y = 1; }
7    z = 2;
8  }
9  assert(x + y + z != 3);
```

Verification of assert(Q):

$$\forall x : P \Rightarrow Q$$

Bug-finding of assert(Q):

$$\exists x : P \land \neg Q$$

line	g	E
0	true	$a \mapsto \alpha, b \mapsto \beta, c \mapsto \gamma$
1	true	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
2	$\neg\alpha$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
5	$\neg\alpha \land \beta \geqslant 5$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
9	$\neg\alpha \land \beta \geqslant 5 \land 0 + 0 + 0 \neq 3$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$

Recap: Soundness and Completeness

- Soundness = "Doesn't lie" or "all claims are true"
- Completeness = "All truths are claimed"
- For Verification (claim is "*program is correct*")
  - Soundness: Reasoning along all possible paths (over-approximation)
- For Bug-Finding (claim is "*a bug exists*")
  - Soundness: Reasoning along feasible paths only (under-approximation)

- Soundness & Completeness is impossible in general (Rice's theorem)
  - Most systems are sound but incomplete (e.g. can't prove all programs, or can't find all bugs)

Recap: Bugs and Reachability

Common trick: convert error case into reachability problem

- assert(p) → if(!p) **ERROR**;
- *x → if(x == NULL) { **ERROR**; } return *x;
- a[i] → if(i < 0 || i > a.length) { **ERROR**; } return a[i];

"Bug finding" is now just about finding inputs that execute every program path

Gotchas: Halting problem and infinite loops

Consider external functions

```
int double (int v) {
    return 2*v;
}

void bar(int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Exercise: Under what path constraints do we hit ERROR?

Consider external functions

```
void bar(int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Consider: What if we could not (or did not want to) analyze the external function?

Consider external functions

```
void bar(int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Consider: What if we could not (or did not want to) analyze the external function?

Consider external functions

```c
int foo(int v) {
    return v*v%50;
}

void baz(int x, int y) {
    z = foo(y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Consider: What if our solver cannot handle non-linear arithmetic or modulo?

Consider external functions

```
int foo(int v) {
    return v*v%50;
}

void baz(int x, int y) {
    z = foo(y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Option 1: Set $\Sigma(z)$ to be a fresh symbolic var

Option 2: Set $\Sigma(z)$ to be a concrete value by ”executing” foo(y) for some y that satisfies path constraint seen so far.

Exercise: How do these options differ in terms of under- or over-approximation?
Recall: soundness/completeness or bug finding or verification

Concolic Execution (= Concrete + Symbolic)

1. Instrument program to collect path constraints during concrete execution (*concrete + symbolic store updates simultaneously*)
2. Run program with concrete inputs (initially random) to collect path constraint *g*
   - Sanity check: Inputs should always be a valid solution to *g*
3. Negate last clause in *g* and solve for model
4. If SAT, then get satisfying assignment as new input and repeat from 2
5. If UNSAT, then pop off last clause and repeat from 3

Concolic Execution: Example

```c
int double (int v) {
    return 2*v;
}

void bar(int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concolic Execution: Example

```
int double (int v) {
    return 2*v;
}

void bar(int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

1. Input: x=0, y=1
   - Path: (2*y != x)
   - Next: (2*y == x) :: **SAT**
2. Input: x=2, y=1
   - Path: (2*y == x) && (x <= y+10)
   - Next: (2*y == x) && (x > y+10) :: **SAT**
3. Input: x=22, y=11
   - Path: (2*y == x) && (x > y+10)
   - **Bug found!!**

Concolic Execution

- **Key advantage**: Always have a concrete input in parallel
- When constraint cannot be modeled (e.g. external function, features not handled by solver), **replace with concrete value**.
- **Soundness**: Concrete replacement is a true under-approximation

Concolic Execution: Example

```
int foo(int v) {
    return v*v%50;
}

void baz(int x, int y) {
    z = foo(y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

1. Input: x=22, y=7
   - Path: (49 != x). // y*y%50 = 49%50 = 49
   - Next: (49 == x) :: **SAT**
2. Input: x=49, y=7
   - Path: (49 == x) && (x > y+10)
   - **Bug found!!**

Concolic Path Condition Soundness

- When is substitution sound?

Concolic Execution: Example

```
int foo(int v) {
    return v*v%50;
}

void baz(int x, int y) {
    z = foo(y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

1. Input: x=0, y=8
   - Path: (14 != x) // y*y%50 = 64%50 = 14
   - Next: (14 == x) :: **SAT**
2. Input: x=14, y=8
   - Path: (14 == x) && (x <= y+10)
   - Next: (14 == x) && (x > y+10) :: **SAT**
3. Input: x=14, y=2
   - Path: (4 != x)
   - **Unsoundness!**

Popular Symbolic/Concolic Tools

- DART (Directed Automated Random Testing)
- CUTE (Concolic Unit Testing Engine)
- KLEE (“dynamic symbolic execution”)
- SAGE (Scalable, Automated, Guided Execution aka ”whitebox fuzzing”)
- Java PathFinder
- Angr
- PyExZ3
- Jalangi